SUPERCHARGE YOUR DATA MODELING

The DynamoDB Book

Analytics Supplement

Alex DeBrie

Analytics with DynamoDB

Alex DeBrie

Version 1.0, 2020-04-03

Table of Contents

Chapter 1. Analytics on your DynamoDB Data

In the data world, there’s a split between *transactional* data workloads and *analytical* data workloads.

Transactional data workloads are fast-acting requests on individual records. As a customer of various tech-enabled applications, your interactions with these applications will be transactional.

Send a tweet. Place an order. View a photo.

These are all transactional access patterns, and they’re where DynamoDB excels. DynamoDB is great at high-scale, low-latency access to individual records. It scales up to terabytes of data while still maintaining single-digit response times.

But DynamoDB isn’t built for the other type of workload. Analytical workloads operate on huge aggregates of your data at any given time. They’re for internal users to ask questions like:

- "What is the top-selling item in North America for the last month?"
- "What’s the week-over-week growth rate for 25 - 34 year old men in Massachusetts?"

These questions are really hard for transactional databases.

If you're optimizing for efficient access of individual records, it's hard to also be really good at large-scale aggregations.

Relational databases can trick you into thinking they're good at both. They certainly have the *language* for both. SQL works well on individual record access as well as writing group-bys and filters on your full table. But those operations will quickly break down as you reach scale.

DynamoDB doesn't even pretend to be good at aggregations. It focuses on nailing the transactional use case, while you need to find a different operation for aggregations.

In this supplement, we'll cover four different options for analytics with your DynamoDB tables:

1. Pre-aggregating your data within DynamoDB;
2. Importing your DynamoDB tables into Amazon Redshift;
3. Exporting your data to Amazon S3 and querying with Amazon Athena;
4. Handling your data in real-time with DynamoDB Streams.

The option you choose will depend on your needs and your team, and you may choose to combine different options for different use cases.

Chapter 2. Pre-aggregations

The first analytics strategy with DynamoDB is the most unique. While the other systems ponder different ways to pull data out of DynamoDB for use in a different system, pre-aggregation uses DynamoDB itself for analytics. This strategy only works for a small number of specific workloads, but it can be a great way to handle these needs without maintaining additional infrastructure.

In this chapter, we'll start by defining pre-aggregation and when you should use it. We'll also discuss some limitations of pre-aggregation. Finally, we'll review some pre-aggregation patterns that you may want to use in your applications.

2.1. What is pre-aggregation?

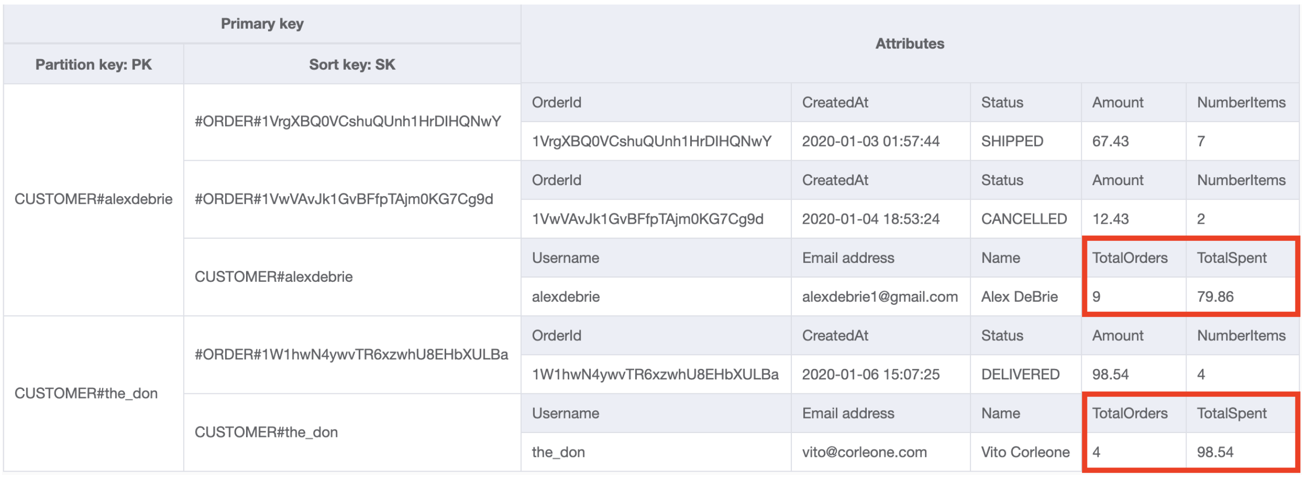
Let's start by defining pre-aggregation. With pre-aggregation, you are aggregating certain metrics *within DynamoDB* to handle some analytics questions you have. You can then build tooling to view this aggregated data by reading from your DynamoDB table.

To take our example, imagine our internal users want to see various aggregated metrics about customers in our system. Perhaps a customer support person wants to view a customer's order history and total amount spent on the site when helping the customer with an issue. Customers with a long history might get a gift certificate to our site after a particularly frustrating experience.

To do this, we build an internal site to view customer data. When an internal user pulls up a customer, it will show information about the customer including how long they've been a customer, their most recent orders, and the total number of orders and amount spent on the site.

To handle this last data, we will keep two attributes on our Customer items: `TotalOrders` and `TotalSpent`. While these properties will never be shown directly to customers, they will be displayed in our internal dashboard.

Our table now looks as follows:



Primary key		Attributes				
Partition key: PK	Sort key: SK					
CUSTOMER#alexdebrie	#ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY	OrderId	CreatedAt	Status	Amount	NumberItems
		1VrgXBQ0VCshuQUnh1HrDIHQNwY	2020-01-03 01:57:44	SHIPPED	67.43	7
	#ORDER#1VwVAvJk1GvBFfpTAjm0KG7Cg9d	OrderId	CreatedAt	Status	Amount	NumberItems
		1VwVAvJk1GvBFfpTAjm0KG7Cg9d	2020-01-04 18:53:24	CANCELLED	12.43	2
	CUSTOMER#alexdebrie	Username	Email address	Name	TotalOrders	TotalSpent
		alexdebrie	alexdebrie1@gmail.com	Alex DeBrie	9	79.86
CUSTOMER#the_don	#ORDER#1W1hwN4ywvTR6xzwhU8EHbXULBa	OrderId	CreatedAt	Status	Amount	NumberItems
		1W1hwN4ywvTR6xzwhU8EHbXULBa	2020-01-06 15:07:25	DELIVERED	98.54	4
	CUSTOMER#the_don	Username	Email address	Name	TotalOrders	TotalSpent
		the_don	vito@corleone.com	Vito Corleone	4	98.54

Notice that both of our Customer items have the `TotalOrders` and `TotalSpent` attributes (outlined in red). When our internal team looks at a Customer in our internal dashboard, it will have this aggregated information without us needing to query all Orders for the Customer to compute this.

Now that we have defined pre-aggregation, let's discuss when you would want to use pre-aggregation.

2.2. When should you use pre-aggregation?

Pre-aggregation works best in the following scenarios:

1. **You know your queries in advance.** With pre-aggregation, you don’t have the flexibility for ad-hoc queries. But if you have specific, targeted questions, you can use pre-aggregation to answer them.
2. **You’re making targeted aggregations about specific items or collections.** Pre-aggregation is great for maintaining counts within a particular parent item or collection of items. It’s not going to work as well for maintaining aggregations across your entire dataset.

The canonical use case for pre-aggregation is the internal dashboard that we showed in the last chapter. In this way, it’s basically just another access pattern for your application. Instead of being a user-facing access pattern, it’s an internally-focused access pattern to show aggregated data to your employees. But the same principles of DynamoDB data modeling still apply: know your access pattern in advance and plan for it accordingly.

Let’s take a look at when pre-aggregation is *not* a good fit.

2.3. Limits of pre-aggregation

Pre-aggregation is pretty limited. As we discussed in the last section, you need to know your aggregation access patterns in advance. For PMs and data analysts that are used to ad-hoc analytic queries, this may not be sufficient for them.

One of the biggest problems with pre-aggregation is that you don't have any ability to change the granularity of your aggregations. For example, imagine you want to show the total number of orders and amount spent for a customer by year. For each customer, you have attributes named `TotalOrders2019`, `TotalOrders2020`, etc. to track the number of orders for each year. You have similar attributes for the amount spent.

While this can show you the yearly numbers for each customer with that information, you have no ability to get a more granular look at that data. What if you want to break down a customer by month in 2020? Or by week? That level of granularity isn't stored in your data, so you're out of luck.

Which leads to the second problem with pre-aggregation: new query patterns are expensive to add. If your analytics data is in a relational database (see the next chapter!), adding a new query pattern is just a slight tweak to your SQL statement. That's not the case here. You'll need to take a developer off other work to update your application logic to add new attributes. If you want to apply this to existing data, you’ll need to write and execute an ETL job to decorate existing items with new attributes. The burden is high enough here that it will make you think twice about asking the questions you want from your data.

Finally, the last problem with pre-aggregation is that it’s hard to do analytics that look across your entire dataset. This pattern shows aggregated information for a particular customer. But what if you want to find the top 10 customers by total spend across your entire system? While you may be able to model for that (see the next section), it’s tricky and won’t work for all applications.

Ultimately, the pre-aggregation approach is limited by the design of DynamoDB. DynamoDB is built for fast lookups of specific items. You can decorate specific items with these aggregated attributes, but you won’t be able to do ad-hoc analysis or efficiently query across your entire dataset.

2.4. Pre-aggregation patterns

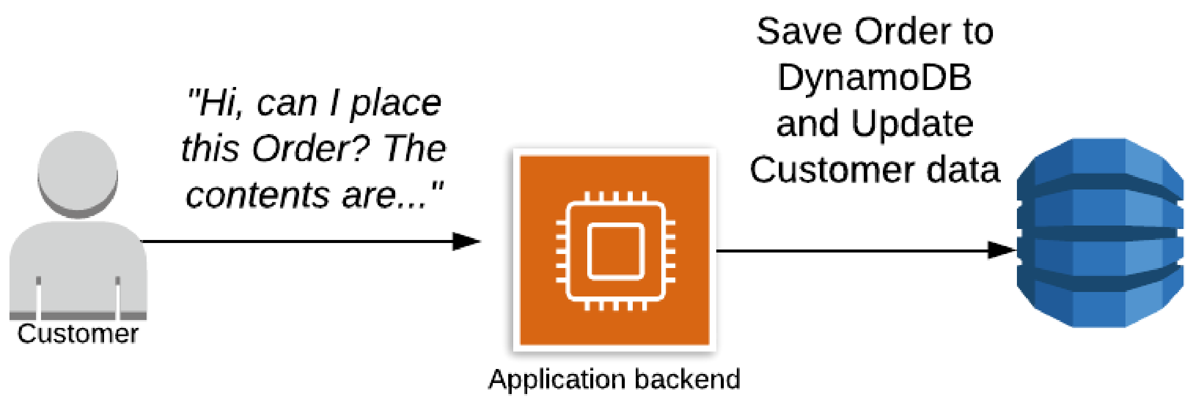
If you’ve decided on the pre-aggregation route for some of your aggregation needs, how do you go about implementing this in your application? In this section, we’ll look at a few different strategies for implementation.

First, we’ll cover whether to write these aggregations in the hot path or asynchronously after the fact. Then, we’ll discuss how to do some 'Top N' queries to query across your entire dataset.

2.4.1. When to write aggregated values

First, let’s discuss when to write your aggregated values. In the example we’ve been using, we want to aggregate the total order number and total amount spent for a particular customer in our application. This means whenever a customer places an order, we need to update the `TotalOrders` and `TotalSpent` attributes on the Customer item.

There are two ways we could do this. First, we could update the Customer item in the 'hot path'. This means updating the Customer item in the same request in which the order is placed. We call this the hot path because it’s an important, user-facing flow that we want to handle quickly and without errors. Slow response times or errors in the order placement flow can result in lost orders for our business!


"Hi, can I place this Order? The contents are..."
Customer
Application backend
Save Order to DynamoDB and Update Customer data

The nice thing about updating the Customer item here is that it’s easier. You don’t have to architecture your application to configure after-the-fact aggregations. Whenever an Order is inserted into our table, the

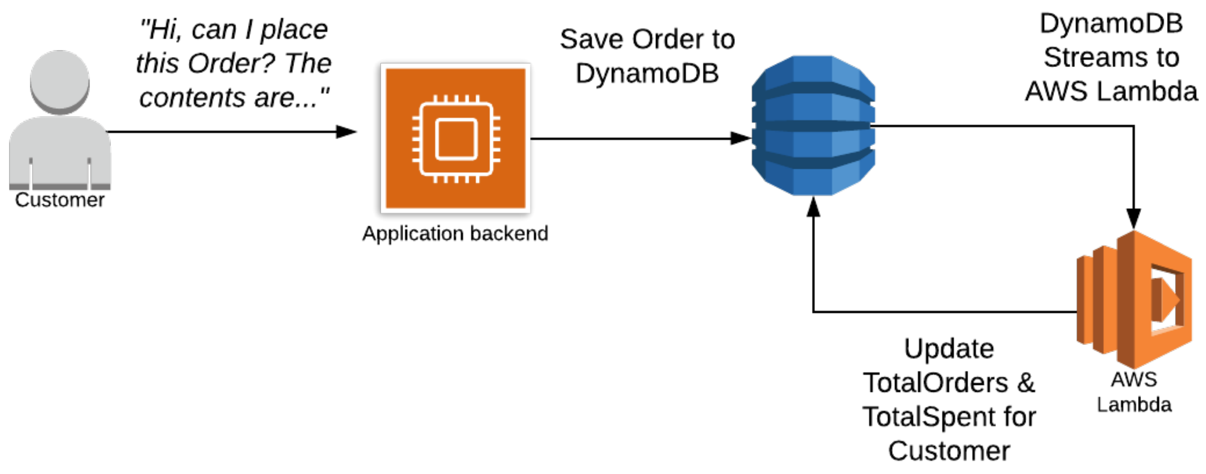
Customer aggregations get updated too.

The downsides are that this adds potential failure modes to our hot path. When we're handling an order, we're already creating an Order item and all the OrderItem items in our table. Now we're adding an additional item with its own potential failure modes to the mix.

What happens if this puts us over the transaction batch size limit? Now we're making additional requests to handle this non-user-facing need.

Or what if our table is throttled because of this additional action? Or what if our update fails because of some ConditionExpression on updating the Customer? Now we're actually *rejecting an order*, which is a huge no-no.

To avoid these issues, we can use a second path to handle these aggregations asynchronously using DynamoDB Streams. While processing our DynamoDB Stream, we can update our Customer item whenever we get notice of a newly-created Order item.


"Hi, can I place this Order? The contents are..."
Customer
Application backend
Save Order to DynamoDB
DynamoDB Streams to AWS Lambda
AWS Lambda
Update TotalOrders & TotalSpent for Customer

This avoids all the hot path issues of writing in the order creation flow. It does add some complexity, as now you're writing logic to consume a DynamoDB Stream, identify new Order items, and update the Customer item accordingly. However, it's likely worth it to take such a risky operation outside of an important request flow.

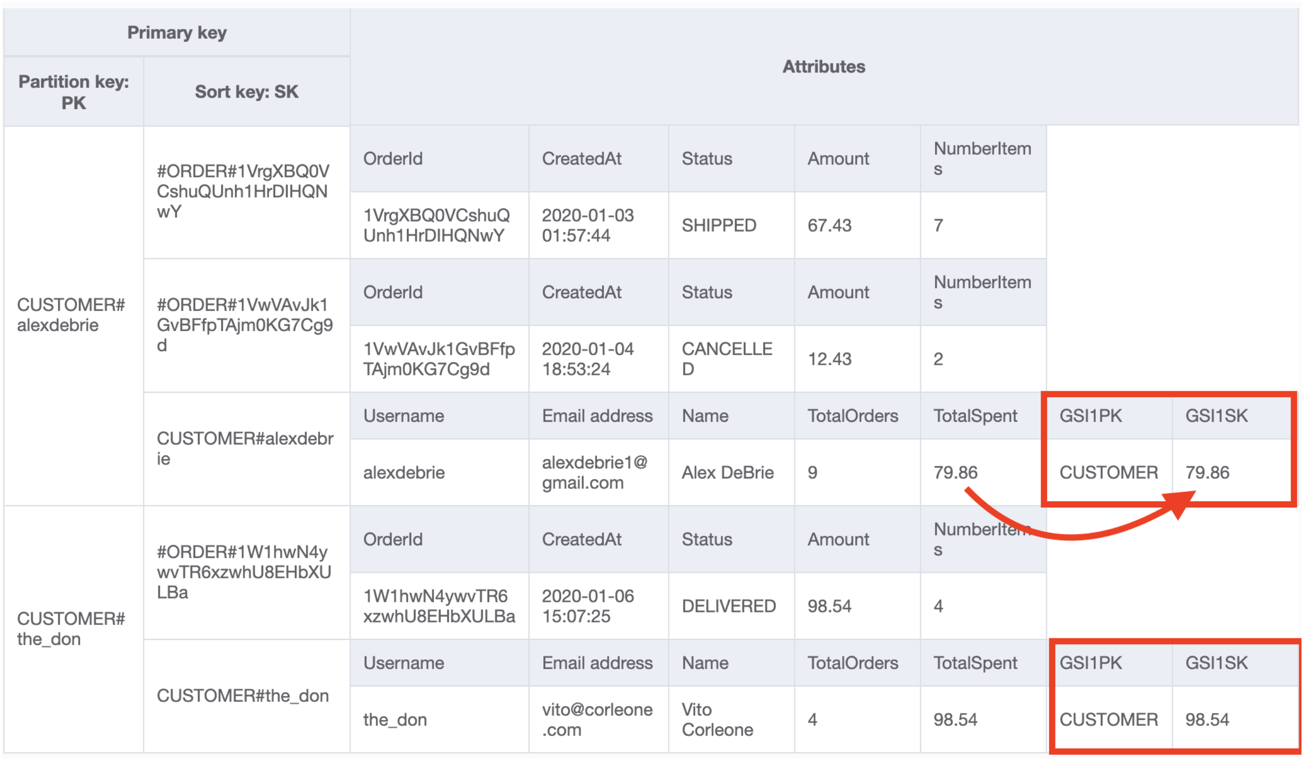
2.4.2. Top N patterns

I mentioned above that it's tough to do "Top N" patterns where you find the top items across your entire dataset. For example, you might want to find the top spending customers across your e-commerce site.

While this is difficult, it's not impossible. Recall that all sorting in DynamoDB is done with the sort key. If you want to order your items based on a particular attribute, you'll need to place them in the same item collection and use your attribute as the sort key.

We can do this with a secondary index. For each Customer item, we'll add a `GSI1PK` attribute whose value is `CUSTOMER` and a `GSI1SK` attribute whose value is equal to the `TotalSpent` attribute.

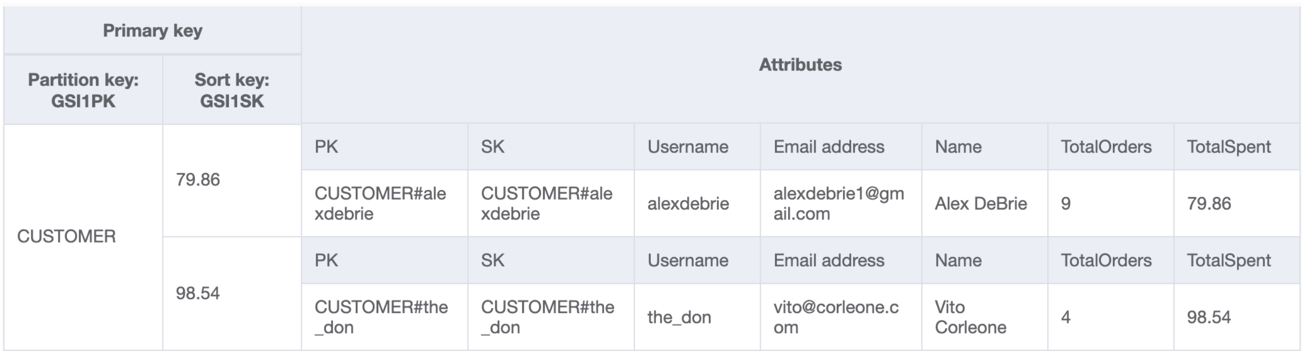
Our table looks as follows:



Primary key		Attributes						
Partition key: PK	Sort key: SK							
CUSTOMER#alexdebrie	#ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY	OrderId	CreatedAt	Status	Amount	NumberItems		
		1VrgXBQ0VCshuQUnh1HrDIHQNwY	2020-01-03 01:57:44	SHIPPED	67.43	7		
	#ORDER#1VwVAvJk1GvBFfpTAjm0KG7Cg9d	OrderId	CreatedAt	Status	Amount	NumberItems		
		1VwVAvJk1GvBFfpTAjm0KG7Cg9d	2020-01-04 18:53:24	CANCELLED	12.43	2		
	CUSTOMER#alexdebrie	Username	Email address	Name	TotalOrders	TotalSpent	GSI1PK	GSI1SK
		alexdebrie	alexdebrie1@gmail.com	Alex DeBrie	9	79.86	CUSTOMER	79.86
CUSTOMER#the_don	#ORDER#1W1hwN4ywvTR6xzwhU8EHbXULBa	OrderId	CreatedAt	Status	Amount	NumberItems		
		1W1hwN4ywvTR6xzwhU8EHbXULBa	2020-01-06 15:07:25	DELIVERED	98.54	4		
	CUSTOMER#the_don	Username	Email address	Name	TotalOrders	TotalSpent	GSI1PK	GSI1SK
		the_don	vito@corleone.com	Vito Corleone	4	98.54	CUSTOMER	98.54

Notice the new attributes outlined in red. Further, the `GSI1SK` value is the same as the `TotalSpent` attribute.

Now when I go to my GSI1, I'll see the following:



Primary key		Attributes						
Partition key: GSI1PK	Sort key: GSI1SK							
CUSTOMER	79.86	PK	SK	Username	Email address	Name	TotalOrders	TotalSpent
		CUSTOMER#alexdebrie	CUSTOMER#alexdebrie	alexdebrie	alexdebrie1@gmail.com	Alex DeBrie	9	79.86
	98.54	PK	SK	Username	Email address	Name	TotalOrders	TotalSpent
		CUSTOMER#the_don	CUSTOMER#the_don	the_don	vito@corleone.com	Vito Corleone	4	98.54

All of my Customer items are in the same partition and sorted by the total amount they've spent on the site. I can do a Query operation against that secondary index with `ScanIndexForward=False` to retrieve the top-spending customers on our site.

The big problem you run into here is with hot partitions. Every single Customer will be placed into the same partition in that secondary index. This means every time you update a Customer item, it's going to do a write to that single partition in our GSI1.

You need to be very careful here. If you don't have enough capacity provisioned in your GSI1 or if you start to exceed the 1000 WCU limit for a single partition, your index updates could lag. If the index replication lags too far, *DynamoDB will block writes on your main table*. This is a very bad situation—you won't be able to take new orders because of your internal aggregation access pattern!

There are a few things you can do to alleviate this issue.

First, you could limit the attributes that are projected into your GSI1 index. If you do a `KEYS_ONLY` projection, then only the primary keys of your table and the secondary index will be written to your GSI1 index. This means updates to other attributes on the Customer item, such as email address, mailing address, or birthdate, won't trigger an update in the GSI1 index.

Second, you could write these Customer items into a different table for aggregation purposes. Because this aggregation is a separate use case from your main application path, it's more acceptable to deviate from single-table design. The downside here is that you are responsible for syncing that data over, but it's better than taking down your production system.

Finally, you could write-shard your data into your GSI1 index. Instead of using a single partition key of `CUSTOMER` for all Customer items, you could use a partition key of `CUSTOMER#<ShardId>`. To get the ShardId for a particular Customer, you would run an operation like the following:

```
shard_id = hash(username) % 15
```

In the code above, I'm hashing the username for the particular Customer, then running a modulo based on the number of shards I want (15). This will return a ShardId between 0 and 14, and I will include that in my GSI1PK value.

This will split writes across multiple shards in my table which will help alleviate concerns about partition write limits. However, it does complicate things at query time. Now when I'm finding the top spending customers in my store, I need to run a Query operation against *each* partition. Once I have those results, then I combine them to find the true top customers.

I recommend against this "scatter-gather" pattern for user-facing operations as it increases response latency. However, for internal use cases like this, it can be a useful strategy.

2.5. Conclusion

In this chapter, we saw how to use pre-aggregation in your DynamoDB tables to handle analytics. Pre-aggregation is just like another data access pattern in your application. You model out your needs and design your table to handle the needs.

While pre-aggregation works for specific, known access patterns, you are limited in flexibility of your analytics queries. In the next chapter, we'll look at importing your data into Amazon Redshift for much more flexible data access.

Chapter 3. Copying your table to Redshift

3.1. Why Redshift?

Redshift is my favorite way to handle analytical queries on my DynamoDB data. There are some limitations, which means it doesn't work well for all situations. However, for situations where it fits, it's the easiest and most powerful query mechanism.

The biggest reason I love using Redshift to query DynamoDB data is because it's so easy. You can run a SQL statement *in Redshift* to copy data from your DynamoDB table directly into your Redshift database. You don't need to perform an ETL process to pull it out of your table, save it to S3, and then load it into Redshift. The Redshift engine will handle everything about getting the data into Redshift.

The second reason I love using Redshift is because it's a relational database that uses SQL. This might be surprising since I talk about the limitations of relational databases and SQL throughout The DynamoDB Book. And I stand by those concerns! But we're in a different situation here.

The limitations of relational databases and SQL is that they start to degrade in OLTP (online transactional processing) performance as your database scales. Joins get slower and queries contend with each other as you’re trying to serve hundreds or thousands of operations per second.

This is less of a concern in an OLAP (online analytical processing) situation, as here. With OLAP workflows, we’re not trying to serve thousands of queries per second as quickly as possible. We’re maybe doing 5-20 concurrent queries that can take a few seconds or even a minute to execute. These are large-scale queries that are doing groupings, aggregations, and sortings over millions or billions of records. Top-end performance is less important.

The flexibility of SQL here is much more useful. Product managers or data analysts can write ad-hoc queries to answer any questions they have about the data. Further, Redshift is based on PostgreSQL 8.4 and thus has compatibility with a lot of Postgres libraries and tools. You can use query interfaces like pgAdmin or hosted data interfaces like Looker and Redash to query and visualize your data.

Finally, Redshift is great as a way to combine multiple data sets. If you have a microservice architecture or if your data is spread out across databases, event logs, and third-party systems, it can be really hard to combine that data together. Redshift makes it easier to ingest data from different sources. Once the data is in Redshift, combining that data is just a join away.

3.2. Why not Redshift?

Now that we know why Redshift is great, let's see when Redshift may not be a good solution for you.

The biggest problem with Redshift is the price. Redshift doesn't have pay-per-use pricing like DynamoDB; you need to pay for an always-on, provisioned cluster. The lowest Redshift instance is $0.25 per hour in the cheapest AWS regions, meaning a Redshift cluster will cost you around $2200 annually. And that's for a fairly small cluster that can hold 260GB of data.

When talking with DynamoDB users, I recommend using Redshift for your DynamoDB data if Redshift is already a part of your company's data strategy. If you're invested in using Redshift as a source of truth for internal analytics, then it makes sense to add DynamoDB to the mix. If you're not using Redshift yet, it can be an expensive leap to spin up a cluster for *just* your DynamoDB data.

The second biggest problem with using Redshift for DynamoDB data is the limited support for DynamoDB data types. If you're using the `COPY` command in Redshift to ingest data from DynamoDB (discussed below), you can only handle attributes of type String or Number. That means if you have Maps, Lists, or Sets, you're out of luck.

At a quick glance, this makes sense. Normalization principles teach us that each column value in a relational schema should be atomic, meaning it cannot be broken down any further. Using a map or a list violates this principle. And the SQL language and relational principles make it difficult to query in a complex column value like that.

But I still wish Redshift would at least let you ingest the data, even if it converted it to a JSON string. You often do transformation of your data *within Redshift* after you load it, and we'll see some examples of that below in the advanced section. It'd be nice if I could load the raw data and then play with it. With the current limitations, I can't ingest complex data types at all. If I need that in my Redshift cluster, I have to do my own ETL work without leaning on the Redshift `COPY` command.

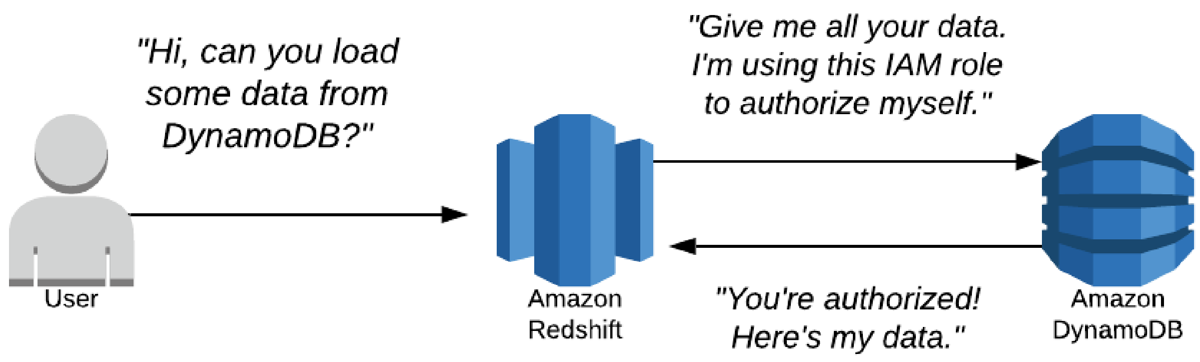
Finally, the last limitation of using Redshift is that it's not real-time. If you're loading into Redshift, you'll be doing it through a batch process where you periodically run a command to ingest the current DynamoDB data into your Redshift cluster. As soon as that command finishes, you'll start falling behind the real data in your application.

This is a reality of *most* internal analytics operations. Real-time is a hard problem, so you often get some batching with a delayed look at data. However, if real-time is a requirement for you, you can read about using DynamoDB Streams in Chapter 5 of this booklet.

3.3. How to ingest DynamoDB data into Redshift

Let’s get into the nuts and bolts of how to actually get your data into Redshift.

At a high level, the process looks as follows:


"Hi, can you load some data from DynamoDB?"
User
Amazon Redshift
"Give me all your data. I'm using this IAM role to authorize myself."
"You're authorized! Here's my data."
Amazon DynamoDB

You will execute a SQL statement in your Redshift cluster. The statement will be a `COPY` statement, indicating you want to copy data from an external source (here, a DynamoDB table) into your Redshift cluster. With Redshift, you can also use the COPY command to ingest data from Amazon S3 or other external sources.

The COPY statement will be similar to the following:

```
COPY ecommerce FROM 'dynamodb://Ecommerce'
IAM_ROLE 'arn:aws:iam::0223456789012:role/MyRedshiftRole'
READRATIO 50
REGION 'us-east-2';
```

Interpreted, this statement says:

1. Copy data into my `ecommerce` table in Redshift
2. From the `Ecommerce` DynamoDB table in my account located in region `us-east-2`
3. Use the IAM role `arn:aws:iam::0223456789012:role/MyRedshiftRole` to read data from my DynamoDB table
4. Use a maximum of 50% of my DynamoDB table's read capacity when performing the Scan.

As Redshift loads data into your table, it will match data from DynamoDB attributes into columns with the same name (case insensitive) in your Redshift table.

Now that we know the general overview of loading data, let's dive into some specifics.

3.3.1. IAM permissions with Redshift Copy

Let's start with IAM permissions in a Redshift copy command. As discussed in the Authorization chapter of the Operations booklet, all access to DynamoDB is controlled via AWS Identity and Access Management (AWS IAM). And this includes access from your Redshift cluster! Your cluster must have the proper permissions to scan your DynamoDB table.

When you create a Redshift cluster, you can associate IAM roles with the cluster. You can also modify your cluster after it's been created to associate more roles.

The CloudFormation to create a Redshift cluster with an

IAM role that can access the `Ecommerce` DynamoDB table is as follows:

```
Resources:
  RedshiftRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2022-10-17
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - redshift.amazonaws.com
            Action:
              - 'sts:AssumeRole'
      Policies:
        - PolicyName: AllowScanEcommerceTable
          PolicyDocument:
            Version: 2022-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'dynamodb:Scan'
                Resource:
                  - 'arn:aws:dynamodb:us-east-2:223456789012:table/Ecommerce'
  RedshiftCluster:
    Type: AWS::Redshift::Cluster
    Properties:
        DBName: "redshift"
        MasterUsername: "master"
        MasterUserPassword: 's4cret'
        NodeType: "ds2.xlarge"
        ClusterType: "single-node"
        IamRoles:
          - Fn::GetAtt:
            - RedshiftRole
            - Arn
```

In this CloudFormation template, we're creating two resources. First, we create an IAM role that will be used by our Redshift cluster. It has a trust policy that states that only the Redshift service may assume the role. It also provides IAM permissions to allow for the `dynamodb:Scan` operation on our Ecommerce table.

Second, we're creating our Redshift cluster. While creating that cluster, we associate our IAM role with the cluster by including the ARN in the `IamRoles` property of the cluster.

Now our Redshift cluster could access our DynamoDB table.

3.3.2. Creating a table

Once you have your Redshift cluster created, you'll need to create a table in Redshift to hold your DynamoDB table data. You will do this by executing a SQL command in your Redshift cluster.

The command to create your table will be similar to the following:

```
CREATE TABLE ecommerce (
    type VARCHAR(200),
    username VARCHAR(200),
    name VARCHAR(200),
    email VARCHAR(200),
    orderId VARCHAR(200),
    address VARCHAR(500),
    createdAt TIMESTAMP,
    status VARCHAR(50),
    totalAmount INTEGER,
    numberItems FLOAT,
    itemId VARCHAR(200),
    price FLOAT,
    amount FLOAT,
    totalCost FLOAT
);
```

With my create table command, I specify all of the attribute names that I want copied across all of my entity types in my DynamoDB table.

A few things to call out here. First, I didn't copy the `Addresses` property from my Customer item in my ecommerce store. That’s because the `Addresses` property is of type Map, which Redshift cannot ingest from a table.

However, I did include the `Address` property from my Order item. The `Address` property is of type String as it’s a *stringifed version of an object*. We’ll be able to use some of Redshift’s JSON parsing functions to extract data from it.

Further, notice that each column has a type. The basic rule is to use `TIMESTAMP` for dates and timestamps, `INTEGER` for integers, `FLOAT` for floats, and then `VARCHAR(length)` for anything else, where 'length' is the maximum length of a value in that column.

3.3.3. Copying data

Once you have your table created, then you need to get data into your table. You can do that with the following SQL command in your Redshift cluster:

```
TRUNCATE TABLE ecommerce;

COPY ecommerce FROM 'dynamodb://Ecommerce'
IAM_ROLE 'arn:aws:iam::0223456789012:role/MyRedshiftRole'
READRATIO 50
REGION 'us-east-2'
TIMEFORMAT 'auto';
```

This is pretty similar to the command we saw earlier, with two additional notes.

First, we’re running a `TRUNCATE TABLE` command at the beginning. The COPY command will append all new rows to your existing table. It’s common to run this COPY command regularly, such as every morning, to get the latest data in your table. However, you need to make sure to delete all existing rows first to ensure you don't have duplicates. The `TRUNCATE TABLE` command will delete all rows from your table.

Second, we added a `TIMEFORMAT 'auto'` parameter. When we have columns of type `TIMESTAMP`, this tells Redshift how to parse the value from our source. Since we're using the standard ISO 8602 format, we can set it to 'auto'. If we had a different format, we could use a time format string to indicate how it should be parsed.

3.4. Advanced topics in Redshift copying

The previous section covered the basics on getting data into Redshift, but you might have more complex needs. I want to cover two additional areas around querying your data in Redshift. First, we'll talk about re-normalizing your data. Then, we'll talk about how to handle permissions when ingesting data from DynamoDB tables in different accounts.

3.4.1. Re-normalization

In The DynamoDB Book, we talk about normalization in the context of a relational database. We also see why denormalization is helpful in DynamoDB.

But now that our data is back into a relational database like

Redshift, we want to *re-normalize* our data to make it more easily queryable. Again, we're not worried about the performance cost of joins here. We're more concerned about the employee efficiency cost of being able to quickly answer ad-hoc questions with our data.

After ingesting a DynamoDB table with a single-table design, all items are in a single table. We need to split those out into separate tables. You can do so with the following SQL command:

```
CREATE TABLE ecommerce_customers AS (
    SELECT username, name, email
    FROM ecommerce
    WHERE type = 'Customer'
);

CREATE TABLE ecommerce_orders AS (
    SELECT username, orderId, address, createdAt, status, totalAmount,
numberItems
    FROM ecommerce
    WHERE type = 'Order'
);

CREATE TABLE ecommerce_order_items AS (
    SELECT orderId, itemId, price, amount, totalCost
    FROM ecommerce
    WHERE type = 'OrderItem'
);
```

In the queries above, I'm using the `CREATE TABLE AS` operation to create a new table from the result of a `SELECT` statement. I'm creating a table for each of my three entity types: Customers, Orders, and OrderItems. In each one, the `SELECT` statement to create the table is selecting only the attributes for that entity type, and it's filtering on the `Type` property that we've added to each item in our table.

Once we've done that, we can answer some really great questions in our data, such as:

Which customers spent the most money in the last 30 days?

```
SELECT username, sum(totalAmount)
FROM ecommerce_orders
WHERE createdAt > GETDATE() - 30
GROUP BY 2
ORDER BY 2 DESC
LIMIT 50;
```

What are the top-selling items of all time?

```
SELECT itemId, sum(totalCost)
FROM ecommerce_order_items
GROUP BY 2
ORDER BY 2 DESC
LIMIT 50;
```

Now your PMs and data analysts love you because they can do their job without writing DynamoDB queries!

3.4.2. Cross-account permissions

It's becoming more and more common to use multiple AWS accounts within a single organization. You might separate your accounts by environment (e.g., dev, staging, prod), by service, or by team.

When you do that, it's possible you'll have a master Redshift cluster in one account that is responsible for ingesting data from all different accounts. This complicates the IAM story around Redshift COPY, but it's still doable. Let's see how.

Imagine we have two AWS accounts. The one with our Redshift cluster is number `211111111111`. The one with our DynamoDB table is number `222222222222`.

First, in our account with our DynamoDB table, we'll create an IAM role that can be *assumed* by the Redshift role in our other account. This role will have permission to scan our DynamoDB table.

```
Resources:
  RedshiftScanRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: RedshiftScanRole
      AssumeRolePolicyDocument:
        Version: 2022-10-17
        Statement:
          - Effect: Allow
            Principal:
              AWS:
                - 'arn:aws:iam::211111111111:role/RedshiftRole'
            Action:
              - 'sts:AssumeRole'
      Policies:
        - PolicyName: AllowScanEcommerceTable
          PolicyDocument:
            Version: 2022-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'dynamodb:Scan'
                Resource:
                  - 'arn:aws:dynamodb:us-east-2:223456789012:table/Ecommerce'
```

We've created a role with the `dynamodb:Scan` permission on our Ecommerce table. We also have a trust relationship that says the role may be assumed by a specific IAM role: the `RedshiftRole` from our Redshift account.

In our Redshift account, we'll create our Redshift cluster and IAM with the following:

```
Resources:
  RedshiftRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2022-10-17
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - redshift.amazonaws.com
            Action:
              - 'sts:AssumeRole'
      Policies:
        - PolicyName: AllowScanEcommerceTable
          PolicyDocument:
            Version: 2022-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'sts:AssumeRole'
                Resource:
                  - 'arn:aws:iam::222222222222:role/RedshiftScanRole'
  RedshiftCluster:
    Type: AWS::Redshift::Cluster
    Properties:
        DBName: "redshift"
        MasterUsername: "master"
        MasterUserPassword: 's4cret'
        NodeType: "ds2.xlarge"
        ClusterType: "single-node"
        IamRoles:
          - Fn::GetAtt:
            - RedshiftRole
            - Arn
```

This is pretty similar to the original CloudFormation statement we had earlier in this chapter. The big difference is that instead of this role having `dynamodb:Scan` permissions directly, this role now has permission to use `sts:AssumeRole` on our role in the other account. As it assumes that role, it can then use the permissions on that role to scan the DynamoDB table.

Finally, let’s look at our SQL statement to handle cross-account access:

```
TRUNCATE TABLE ecommerce;

COPY ecommerce FROM 'dynamodb://Ecommerce'
IAM_ROLE
'arn:aws:iam::211111111111:role/RedshiftRole,arn:aws:iam::222222222222:role/R
edshiftScanRole'
READRATIO 50
REGION 'us-east-2'
TIMEFORMAT 'auto';
```

Notice that the `IAM_ROLE` property now has two IAM role ARNs separated by a comma. We're using *role chaining* to handle this. Our Redshift cluster can't assume the cross-account role directly, so it first assumes the role associated with the table, then uses that role's permissions to assume the role in our other account. With those permissions, it can scan the Ecommerce table.

This IAM role-chaining is a powerful way to give cross-account access to just your Redshift cluster without allowing broader access to users and service in your Redshift account.

3.5. Conclusion

In this chapter, we saw how and why to use Redshift for analytics with your DynamoDB table. Redshift is the easiest way to get going with analytics as long as you can afford it and you're not using complex attribute types in DynamoDB.

After we saw why to use it, we did some walkthroughs on how to get your DynamoDB tables into Redshift. We saw how to set up permissions, create tables, and copy data. Then, we saw some advanced tips like re-normalizing your data and handling cross-account permissions.

In the next chapter, we’ll see how to get powerful SQL queries on your DynamoDB in situations where Redshift doesn’t work for you.

Chapter 4. Data Exports with AWS Glue and Amazon Athena

Redshift is pretty powerful, but it can be expensive. You’re paying for an always-on database with terabytes of storage. You’re already using one of those with DynamoDB—do you really want to pay for another one?

In this chapter, we’ll talk about Amazon Athena. Athena is a more 'serverless' option to data analytics. Rather than paying for an always-on machine, you only pay for the queries you make.

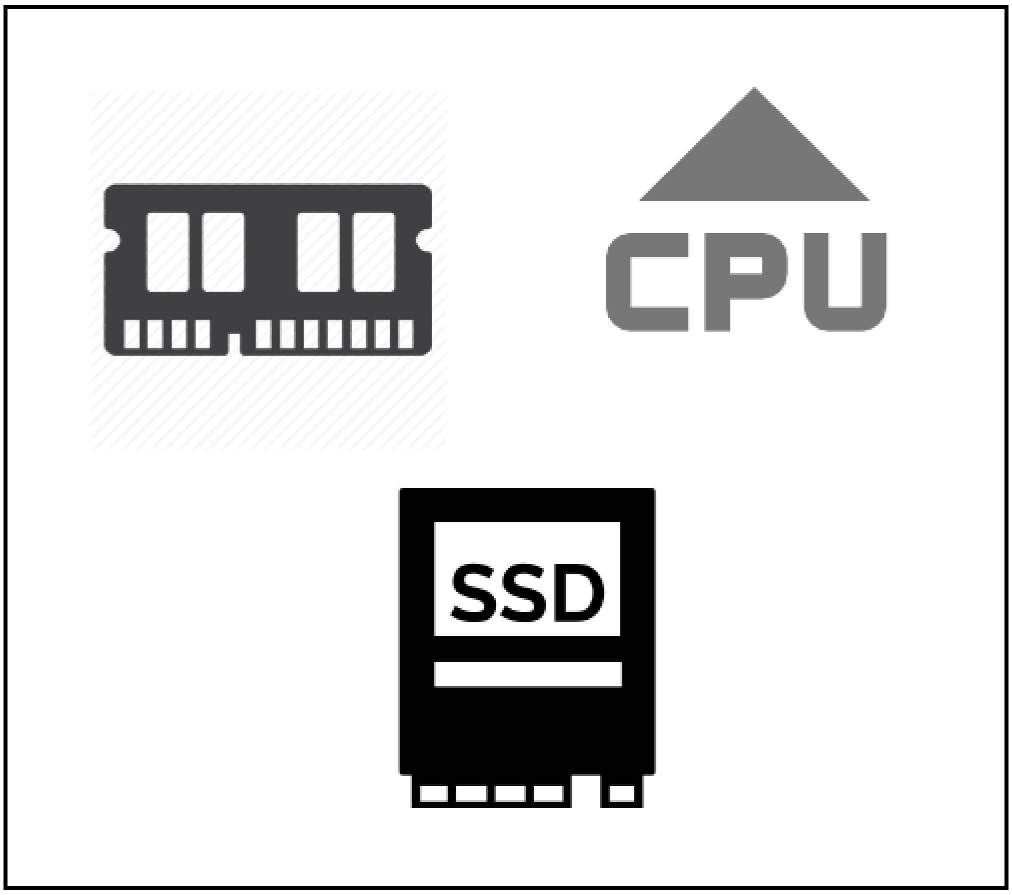
As we discuss Athena, we’ll cover what it is and when it’s a good fit for your analytics workloads. We’ll also talk about AWS Glue, a partner product to Athena that helps with exporting data from your DynamoDB table. Next, we’ll discuss when Athena and Glue might not be a good fit for you. Finally, we’ll cover some advanced topics about working with Athena like IAM permissions, partitioning, and data formats.

Here we go.

4.1. What is Amazon Athena?

Amazon Athena is a hosted query execution engine that operates on objects in Amazon S3. It allows you to run SQL queries against structured data in S3.

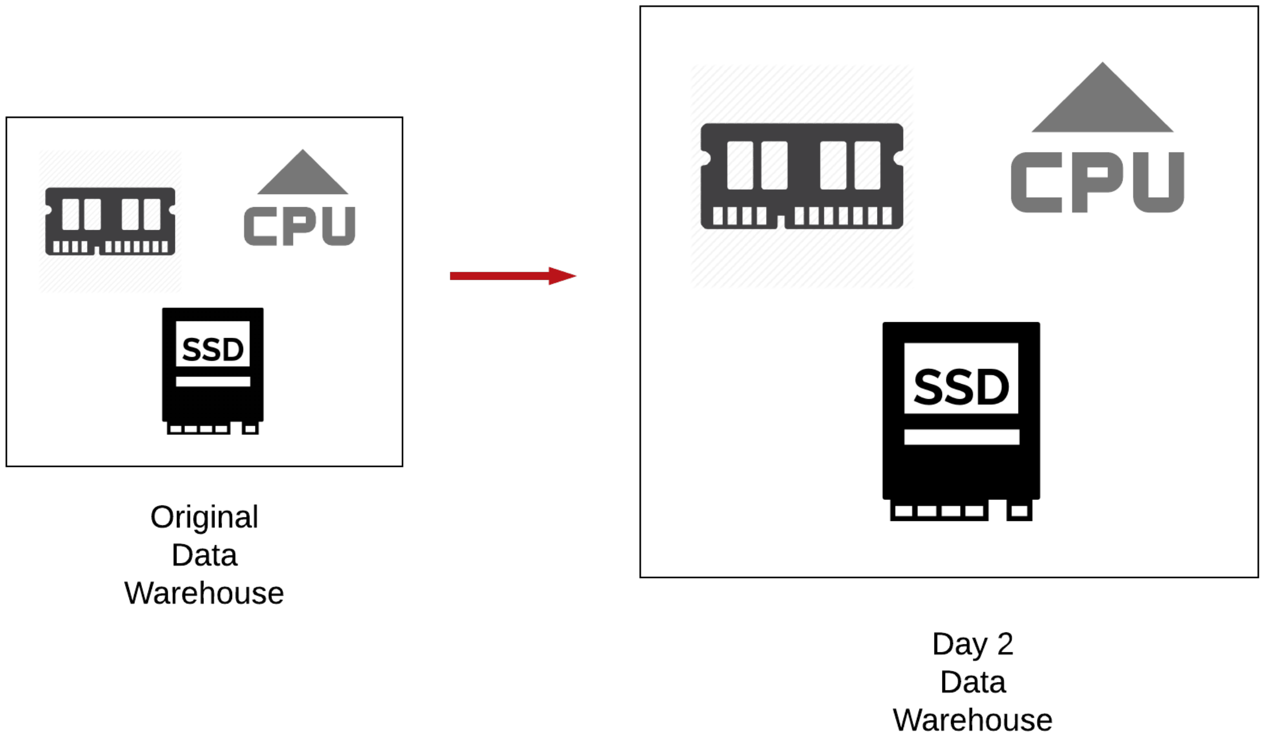
This is pretty amazing. Think about a traditional database. You usually have an instance with some amount of compute and memory attached to some disks used for storage. When you send write data to the database, it organizes it as needed on the disks.



Original
Data
Warehouse

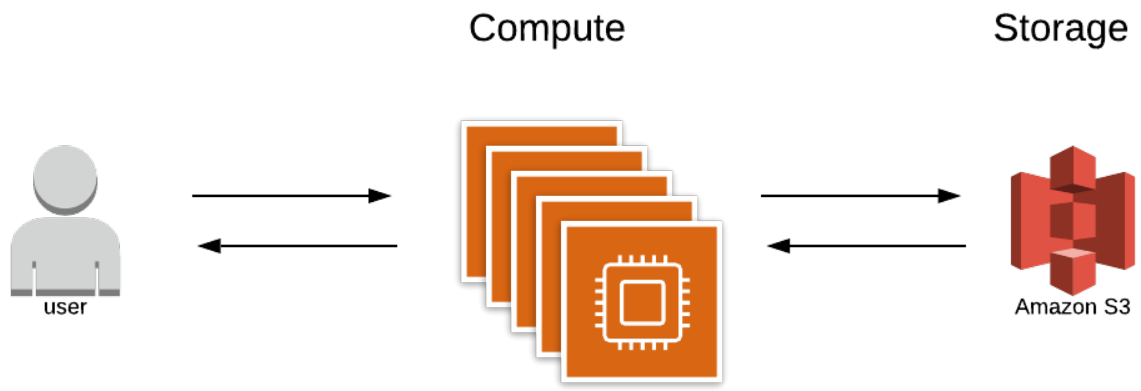
When you send a read query, it uses the CPU and RAM to read from disk, sort and filter the rows, and then return the results to you.

However, as your data set gets bigger or as your query needs get larger, it's harder and harder to scale your database. You need more disk to hold your data, and you need a bigger instance with more CPU and RAM to process it. This need is particularly acute with data warehouses that are storing and processing enormous amounts of data.



Athena takes a different approach. Rather than combine this on one machine, it decouples into constituent parts. Instead of an attached disk for storage, it uses Amazon S3, which is an infinitely-scalable, pay-per-use storage service with great availability and durability characteristics.

Instead of CPU and RAM that's local to the disk, it uses independent worker nodes to process a query. These worker nodes can work in parallel to read and perform initial processing.



After doing the initial processing, results from the various workers can be combined to handle final filtering, sorting, and presentation. Like the storage component, the number of workers can be scaled up or down depending on the query size.

This provides a lot of flexibility around data warehousing. Instead of having an always-on cluster like Amazon Redshift, you only pay for the amount of data processed in Athena (currently $5 per TB in the cheaper regions).

Athena is a good choice over Amazon Redshift if you like the pay-per-use pricing model. It's also a good choice if you have complex data types, like maps, lists, and sets, that cannot be ingested directly into Amazon Redshift.

4.2. What is AWS Glue?

AWS Glue is a managed ETL service provided by AWS. ETL stands for 'extract, transform, and load' and is generally used to describe large-scale data processing workflows. Often you will pull data out of a production data source ('extract'), manipulate or enrich the extracted data ('transform'), and store in a place that can be used for large-scale data analytics ('load'). AWS Glue helps with this procedure.

I mentioned in the previous section that Amazon Athena queries data from S3. If you want to use Athena with DynamoDB, you need to find a way to export your data into S3. AWS Glue is the easiest way to handle that.

With AWS Glue, you create a *job* to export data from a data source to a data target. Our data source will be our DynamoDB table, and the data target will be an S3 bucket. AWS Glue will pre-generate the job code to handle the transformation, and you can modify it as needed to transform your data into a SQL-friendly format.

4.3. Why not Athena and Glue?

Now that we know a bit about Athena and Glue and why we might choose these technologies, let’s discuss when Athena and Glue might not be a good fit.

First, queries to Amazon Athena will not be as fast as equivalent queries in an always-on data warehouse like Amazon Redshift. Because there is more overhead in preparing compute, reading from S3, and grouping results, it's naturally going to take more time with a decoupled system like Athena. That said, your data warehouse is usually not there for fast queries. Even Redshift will take a few seconds for most queries. The difference here is maybe going from 3 seconds to 15 seconds. It's probably not a show-stopper for you.

A second reason you might avoid Athena is that certain SQL patterns will be much less performant in Athena than in Redshift. Athena does support joins, but large-scale joins will be inefficient as it will need to scan entire tables to find matching rows. Further, if you have large grouping and ordering requirements, Athena will need to push those all to a single worker at some point to get consistent results. This can slow down your queries if you have a large result set.

Finally, you might want to avoid using Glue and Athena because it complicates the ETL work. With the Redshift COPY command, it was a simple SQL statement to pull all of your data into Redshift. If you need to split your data into different tables, it was a pretty straight-forward SQL query to do so. To do the equivalent with Glue, you would need to mess with the transformation script to split your data into different buckets or folders. It's not impossible, but it does increase your workload.

4.4. Advanced topics in Athena

As you get deeper into Amazon Athena, you'll realize there are a number of tricks to use Athena effectively. There are a few advanced topics I want to cover as you start working with Athena:

1. Handling permissions in Athena
2. Creating new tables with the Create Table As Command
3. Partitioning your data for better query performance
4. Data formats and compression

4.4.1. Permissions in Athena

When executing a query in Athena, the Athena service will have the permissions *of the entity that executed the query*. This means if you execute the query as an IAM user, you will have the permissions of that user. If you have assumed a role to execute a query, you will have the permissions of that IAM role.

Note that this is different than the role assumption method that Redshift uses. You cannot associate a role with the Athena service that will be used to perform actions.

A good approach here would be to have an IAM role that must be assumed to perform any Athena queries. Then, you can give permission to various IAM users to assume that role. This way all permissions would go through that specific IAM role rather than being spread across multiple users.

The CloudFormation example might look like this:

```
Resources:
  AthenaRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: AthenaRole
      AssumeRolePolicyDocument:
        Version: 2022-10-17
        Statement:
          - Effect: Allow
            Principal:
              AWS: "arn:aws:iam::123456789012:root"
            Action:
              - 'sts:AssumeRole'
      Policies:
        - PolicyName: AllowScanEcommerceTable
          PolicyDocument:
            Version: 2022-10-17
            Statement:
              - Effect: Allow
                Action:
                  - "athena:*"
                Resource:
                  - "*"
              - Effect: Allow
                Action:
                  - "s3:GetBucketLocation"
                  - "s3:GetObject"
                  - "s3:ListBucket"
                  - "s3:ListBucketMultipartUploads"
                  - "s3:ListMultipartUploadParts"
                  - "s3:AbortMultipartUpload"
                  - "s3:PutObject"
                Resource:
                  - "arn:aws:s3:::ecommerce-table-bucket",
                  - "arn:aws:s3:::ecommerce-table-bucket/*"
  AllowAssumeAthenaRolePolicy:
    Type: AWS::IAM::Policy
    Properties:
      PolicyName: AllowAssumeAthenaRole
      Statement:
        - Effect: Allow
          Action:
            - 'sts:AssumeRole'
          Resource:
            - Fn::GetAtt:
              - AthenaRole
              - Arn
```

In the CloudFormation above, we’re creating two resources. First, we’re creating a Role that can be used with AWS Athena to use the Athena service and have permissions for certain S3 actions on the bucket used to hold my DynamoDB data exports. Then, we create an IAM policy that gives permission to assume the role. We would attach this policy to specific users or to a group that includes users that we want to have Athena access.

Because there is no role assumption in Athena directly, it means we also cannot do role chaining for cross-account access like we did in Redshift. If you want to do cross-account access for your S3 buckets, you would use an S3 bucket policy. A bucket policy has similar syntax to an IAM statement but it is attached directly to an S3 bucket. It can be used to control who has access to the bucket.

The S3 bucket policy might look as follows:

```
{
   "Version": "2012-10-17",
   "Id": "CrossAccountAthena",
   "Statement": [
      {
          "Sid": "MyStatementSid",
          "Effect": "Allow",
          "Principal": {
             "AWS": "arn:aws:iam::123456789123:role/AthenaRole"
          },
          "Action": [
             "s3:GetBucketLocation",
             "s3:GetObject",
             "s3:ListBucket",
             "s3:ListBucketMultipartUploads",
             "s3:ListMultipartUploadParts",
             "s3:AbortMultipartUpload",
             "s3:PutObject"
          ],
          "Resource": [
             "arn:aws:s3:::my-cross-account-bucket",
             "arn:aws:s3:::my-cross-account-bucket/*"
          ]
       }
    ]
 }
```

Note that this policy applies to a specific Principal—the IAM role that we created in our other account. Thus, if that IAM role tries to query this bucket, it will have the permissions stated in this bucket policy.

4.4.2. Create Table As

In the previous chapter about Redshift, we mentioned you often need to do some ETL work after importing your DynamoDB table into Redshift. The same principle is true here.

With Athena, you can reshape your data using the `CREATE TABLE AS` and `INSERT INTO` SQL statements. Let's explore each of these.

The `CREATE TABLE AS` statement is just like how it worked in Redshift—you can create a new table as the result of a SQL `SELECT` statement. However, with Athena, there's no disk to write to like in Redshift. Rather, it writes to a new location in S3.

Modifying our SQL statement from the last chapter to work with Athena looks as follows:

```
CREATE table ecommerce_customers
WITH (
  format='PARQUET',
  parquet_compression='SNAPPY',
  external_location = 's3://ecommerce-table-bucket/etl/customers'
)
AS
SELECT username,
       name,
       email
FROM ecommerce
WHERE type = 'Customer'
```

The statement is pretty similar to the Redshift query with some small tweaks. You need to describe where you want to place the data (`external_location`) as well as details on the format and compression of your data.

The format of your data can make a big difference with Athena queries. See the section below on data formats for tips on proper data formats.

4.4.3. Partitioning your data

With Athena, you're charged based on the amount of data read. This can be expensive if you need to query your *entire* dataset every time even if you're adding some filters on your data.

To help with this, Athena has the concept of *partitioning* your data. When you write data to S3, you can partition your data in different "folders" according to common filters. For example, if you often filter by time, you may partition your data as follows:

```
data/year=2020/month=01/date=30/...
data/year=2020/month=01/date=31/...
data/year=2020/month=02/date=01/...
data/year=2020/month=02/date=02/...
```

Notice that the key names include `year={year>}`, `month={month}` and `date={date}`. When you create your table in Athena, your Create Table query may look as follows:

```
CREATE EXTERNAL TABLE ecommerce (
    type string,
    username string,
    name string,
    orderId string,
    totalAmount float,
    numberItems string,
    itemId string,
    price float,
    amount integer,
    createdAt timestamp
PARTITIONED BY (year string, month string, date string)
STORED AS 'parquet'
LOCATION 's3://ecommerce-table-bucket/data/' ;
```

Now when you execute SQL queries that filter on `year`, `month`, and `date` in the WHERE clause, Athena knows to skip the S3 objects that are outside of those partitions. This will reduce the amount of data stored.

You'll need to do a little work to massage the data into this partitioned format, but it can make a big difference in your Athena bill if you're making large queries with common filters. The `CREATE TABLE AS` statement greatly simplifies the ETL work that you need to do to partition your data.

4.4.4. Data format and compression

As we just discussed, Athena charges you based on the amount of data read from S3. Partitioning is one way to reduce this cost. Another way is to use proper data formats and compression.

Athena supports a number of different data formats for your data. You can use 'schema-on-read' formats like JSON or you can use 'schema-on-write' formats like Avro, Orc, or Parquet.

The schema-on-read formats like JSON might *feel* easier. You don't need to think about making a schema for your data and ensuring your records conform to it. You can add new properties as needed without updating the schema.

But this is going to be more expensive in a number of ways. First, JSON is much more verbose than the other formats. You need to store all the property names for each record within the record itself. Further, it's harder to organize the data in a columnar way that allows for faster aggregations on large data sets.

The other formats are more suited to large-scale data processing. Because they have a defined schema, that schema can be written once at the top of a file and applied to each record in the file. This allows for a much smaller file size.

Further, formats like Orc and Parquet are designed to work with columnar workloads like large-scale data processing. This means rather than storing each record together with each of its attributes, Orc or Parquet might store each *attribute* together. If I'm trying to find the total amount spent by customer, this can save a lot of time as I only need to read the `username` and `totalAmount` properties on the Order records rather than reading all of the other properties.

Finally, the Parquet file format supports writing metadata about the file at the top of a file. This will include information such as the range of values for various attributes contained in the file. Athena can read just the metadata portion of the file to check if it needs to read the entire file. For example, imagine one of your Parquet files only had customer orders that were created between January 1, 2020 and February 12, 2020. If your query was looking for Orders in 2019, it could skip reading the full contents of this file entirely.

Athena also supports a number of compression options so that your data file sizes are even smaller. You don't need to compress files in the Orc or Parquet format, as they are compressed by default. If you have JSON or flat files, you can use compression algorithms like Snappy, LZO, or gzip to reduce your file size.

4.5. Conclusion

With tools like AWS Glue and Amazon Athena, it's easier than ever to maintain a fast data warehouse without paying for always-on capacity. You can export your raw data with AWS Glue, reshape it with Athena's `CREATE TABLE AS` functionality, and query it within Athena.

This is the option I recommend for most people, especially if the Redshift option doesn't work for you due to cost or your use of complex attribute types in DynamoDB. It's slightly higher maintenance work than Redshift but still pretty low overall.

In the next chapter, we'll look at DynamoDB Streams for those who have a need for more real-time analysis.

Chapter 5. Realtime analytics with DynamoDB Streams

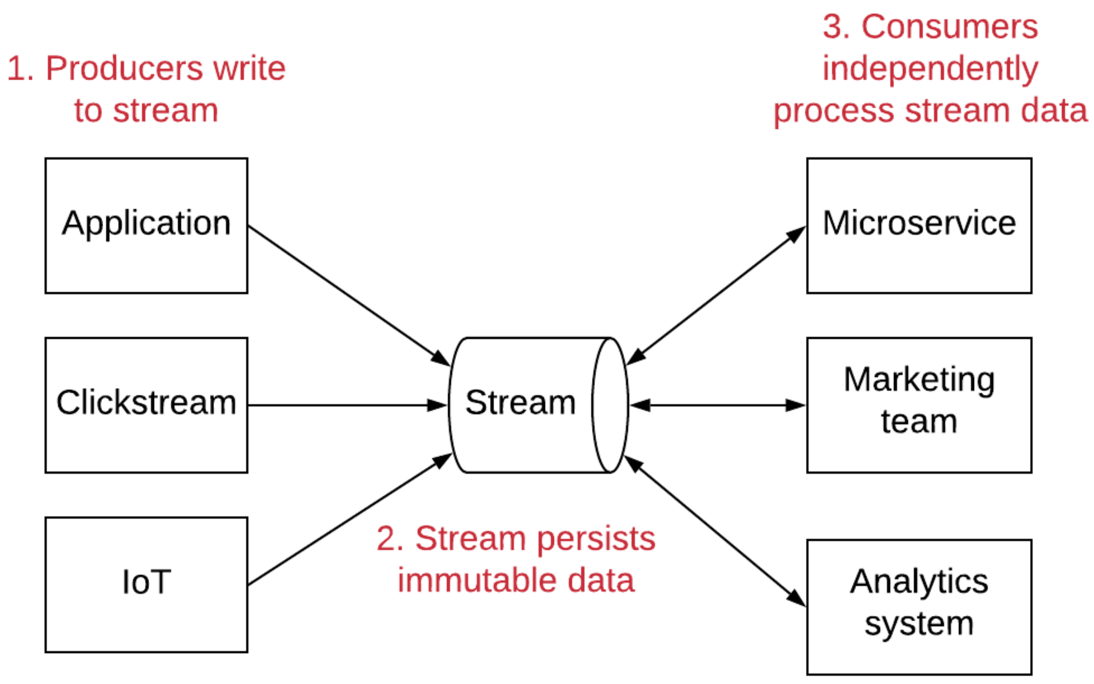
The last analytics pattern is to use DynamoDB Streams to give you near-real-time analytics on your DynamoDB tables. This is the most complex option operationally, but it also gives you the fastest look at your data. If you need up-to-date data for your analytics, this is the option for you.

In this chapter, we'll learn what DynamoDB Streams are and how to use them in analytics systems. We'll also discuss some problems you may run into by building your analytics around DynamoDB Streams. Then, we'll discuss choosing a data store with DynamoDB Streams. Finally, we'll wrap up with some notes on using DynamoDB Streams with AWS Lambda.

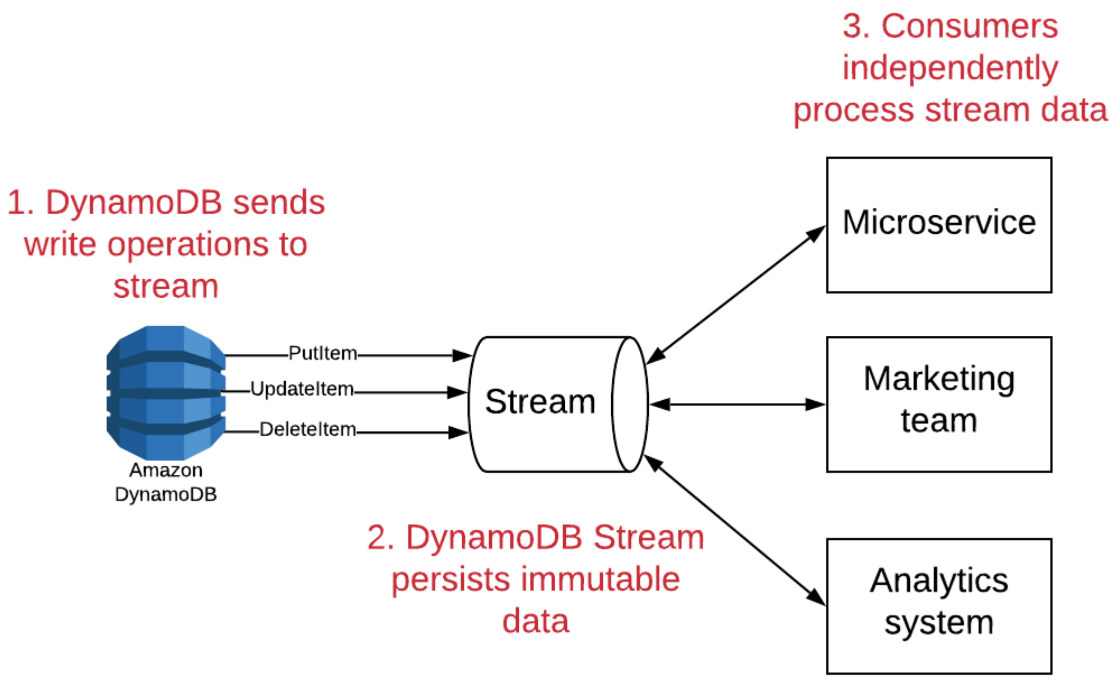
5.1. What and why of using DynamoDB Streams

First, let's discuss what DynamoDB Streams are and how you would use them for analytics.

The concept of streams and event streaming has exploded in popularity over the past decade with the rise of tools like Apache Kafta and Amazon Kinesis. Streams are an immutable sequence of records that can be processed by multiple, independent consumers. The combination of immutability plus multiple consumers have propelled the use of streams as a way to asynchronously share data across multiple systems.


1. Producers write to stream
Application
Clickstream
IoT
Stream
2. Stream persists immutable data
3. Consumers independently process stream data
Microservice
Marketing team
Analytics system

With DynamoDB streams, you can create a stream of data that includes a record of each change to an item in your table. Whenever an item is written, updated, or deleted, a record containing the details of that record will be written to your DynamoDB stream. You can then process this stream with AWS Lambda or other compute infrastructure.


3. Consumers independently process stream data
1. DynamoDB sends write operations to stream
Microservice
PutItem
UpdateItem
DeleteItem
Stream
Marketing team
Amazon DynamoDB
2. DynamoDB Stream persists immutable data
Analytics system

DynamoDB Streams can be used for a few different purposes. Some use them to enable communication of events across microservices. One service can publish the events that occurred in that service—new Order placed; Order status updated; Customer created—and other services can use that to update their data stores. For example, the inventory service might use the OrderPlaced event to adjust inventory levels accordingly, while the marketing service might use the CustomerCreated event to send out welcome emails or look up demographic information about the new Customer.

But for this chapter, we're going to talk about how DynamoDB Streams can enable your analytics workflows. The last two chapters on Redshift and Athena have dealt with batch-based ETL workflows. You periodically export all of the data from your DynamoDB table into a system that can run analytical queries. With DynamoDB Streams, you are getting a continuous feed of actions that are happening in your DynamoDB table. You can use this to enable more real-time analysis.

That said, using DynamoDB Streams is probably the most complex mechanism for data analytics with DynamoDB. Rather than using simple SQL statements to import data into Redshift or off-the-shelf tools like AWS Glue to export your data to S3, you'll need to write actual code to process your DynamoDB Streams. This involves parsing the record given to you, potentially re-normalizing your data in your processing code, and understanding which operations you need to execute in your analytics system.

Further, it can be difficult to merge the stream processing world with the analytical processing world. Stream processing operates on a small number of records continuously fed to your processors. In contrast, most analytical systems like Redshift or Athena are loaded with a large number of records on a small basis. We'll discuss this further in the next section.

5.2. Choosing an analytics datastore with DynamoDB Streams

One of the biggest problems with using streams with analytical systems is figuring out how to get the streaming data into your analytical systems. Redshift can easily ingest hundreds of GBs of data from S3 or DynamoDB in a single command. However, it is not great at small, individual operations like "Delete Customer XYZ" or "Change the status of Order 123 to 'Placed'".

In the sections below, we'll look at three different datastore options. The one you choose will depend on your analytics requirements.

5.2.1. Standard relational database (MySQL, PostgreSQL)

The first option is to use a standard relational database like MySQL or PostgreSQL for your data. There are a few reasons you might want to choose these options.

First, a standard relational database is great at handling lots of individual insert, update, and delete operations. It can work at the same speed as your DynamoDB table and its associated DynamoDB Stream.

Second, a standard relational database will give you ad-hoc query flexibility that you can't get from DynamoDB. You can use the powerful filtering, joining, and sorting features and the SQL language to ask any questions you want of your data.

There is a major downside of using a standard relational database. Relational databases are built for OLTP-like workloads, operating on a small set of items at a time. They're not great at giant aggregation queries, such as "Find me the top spending Customers for all of 2019." That requires scanning the entire Orders table, grouping by CustomerId, and ordering the intermediate results. This is an operation for which standard relational databases are not optimized.

5.2.2. Massively parallel processing (MPP) databases like Redshift or Snowflake

The second option for analytics datastores with DynamoDB Streams is an MPP database (which stands for massively parallel processing). This includes options like Amazon Redshift, which we discussed before, or Snowflake, a popular third-party database.

MPP databases have similar query syntax to standard relational databases. In fact, Amazon Redshift is a fork of a prior version of PostgreSQL! However, the way they store and process data is very different.

In a standard relational database, an individual *row* is stored together. Most operations happen on individual rows, so it makes sense to include all column values for that row together for quick operations. However, this means that analytics queries need to read all values for all rows to perform aggregations.

In contrast, MPP databases use a columnar storage format where *columns* are stored together. For example, in our Orders table, all of the values for `TotalAmount` would be stored together, and the values for `NumberItems` would be stored together. If I run a query for sum the `TotalAmount` by CustomerId, it only needs to read two columns—TotalAmount and CustomerId—to compute my results.

Further, these columnar storage formats use various compression and metadata tricks like we discussed with Athena. Different blocks of storage can indicate which values they have for a particular column, allowing the query engine to skip reading blocks that don't match the given filter for a query.

This columnar storage format makes large aggregations fast, but it makes individual inserts comparably slow. It might take 10 seconds to calculate the most popular order items across a 500GB table, but it might take 3 seconds to retrieve an individual Order by its OrderId. In contrast, a standard relational database could fetch the individual Order row in milliseconds but would spend minutes or hours trying to find the top order items in a 500GB table.

The main takeaway here is that you can't directly update individual records in Redshift or Snowflake from your DynamoDB Streams processor—you simply won't be able to keep up. You'll need to find some way to batch your updates to your database.

One option here is to use Amazon Kinesis Data Firehose to batch up your data. With Kinesis Firehose, you can write individual bits of data from multiple consumers. Kinesis Firehose will buffer that data and occasionally flush it to S3 in a larger chunk. Thus, you'll end up with 128MB chunks of data on S3 that can be ingested more efficiently into

Redshift.

However, your work isn't done once the data arrives in Redshift. At that point, you'll have a table full of Insert, Update, and Delete operations from DynamoDB. You'll need to do the work to merge these individual operations with your existing data. And this is trickier than it sounds!

You'll need to do some combination of the following actions:

1. Load your DynamoDB Stream data into a staging table containing the raw operations. This will include the operation type (Insert, Update, or Delete) as well as the full record after the operation.
2. Using a window function, copy the operations staging data into another table that contains the *most recent* operation for a particular record. For example, if you had a DynamoDB item that was created, updated, and deleted within the period you're buffering data in S3 and your staging table, you only care about the last operation, as that shows the most recent state of your item.
3. With your most recent staging table, update your actual tables with the operations. For items that were deleted, you need to delete the corresponding records. For items that were inserted or updated, you need to replace the existing records with the new version from your operation.

This process can be tricky to get right, and your data may start to drift from your original source over time. You may find that you want to perform a periodic full export & refresh of your table in Redshift and use this incremental strategy to provide a 'pretty close' view of your data between refreshes.

5.2.3. Storage-driven systems like Athena

The last approach is to use a storage-driven system like Amazon Athena. Remember from the last chapter that Amazon Athena separates storage and compute by using Amazon S3 as the storage layer for all data operations.

You can use storage-driven systems in the same way we just saw with MPP systems like Amazon Redshift. All the basic operations like staging areas and joins will work, and you can use the 'CREATE TABLE AS' operation to create a table from a query.

However, these operations are going to be less efficient with Athena. The join operation in a storage-based system is not quite as efficient as in Redshift where you can use the sort key for more efficient joins. Further, there’s no concept of updating rows in Athena. All updates would need to write to a brand new location on S3. This can result in a lot of churn and operations costs as you’re re-writing your entire dataset to a new location each time you merge in a new batch of records.

I find that Athena works best with DynamoDB Streams in 'write-many, update never' type of workloads. We used an example of an IoT application in various parts of The DynamoDB Book, and that's a great fit here. You rarely go back and edit the value for a particular sensor reading in an IoT application. Because of that, we can stream all of our new records directly into S3 for querying with Athena. We can even use the partitioning strategy discussed in the last chapter to partition these records by date. Because we never update records, we don't have to worry about costly joins and rewrites of our table as records change.

5.3. Using Streams with AWS Lambda

There are a few ways to process DynamoDB Streams. You can manually process the stream using the low-level `GetShardIterator` and `GetRecords` API actions. I don't prefer to use this method as it requires managing more of your own infrastructure as well as maintaining state around your location in the stream.

Rather, I prefer to use AWS Lambda to process DynamoDB Streams. This provides a fully-managed experience where you only write the application code to process the stream. AWS will handle subscribing to your stream, providing enough consumers to process the stream, and maintaining state about location in the stream.

If you look at the Big Time Deals example code, I have an example of stream processing related to sending messages about new deals to Brand Watchers and Category Watchers. I use the Serverless Framework to manage my

AWS Lambda deployments, and an example `serverless.yml` file to process a DynamoDB Stream would look as follows:

```yaml
service: big-time-deals-node

provider:
  name: aws
  runtime: nodejs12.x
  stage: prod
  region: us-east-1
  iamRoleStatements:
    - Effect: "Allow"
      Action:
        - "dynamodb:DescribeStream"
        - "dynamodb:GetRecords"
        - "dynamodb:GetShardIterator"
        - "dynamodb:ListStreams"
      Resource:
        - Fn::Join:
          - "/"
          - - Fn::GetAtt:
                - BigTimeDealsTable
                - Arn
            - "stream/*"

functions:
  streamHandler:
    handler: handlers/streamHandler.handler
    events:
      - stream:
          type: dynamodb
          arn:
            Fn::GetAtt: [BigTimeDealsTable, StreamArn]


resources:
  Resources:
    BigTimeDealsTable:
      Type: AWS::DynamoDB::Table
      Properties:
        AttributeDefinitions:
          - AttributeName: "PK"
            AttributeType: "S"
          - AttributeName: "SK"
            AttributeType: "S"
        KeySchema:
          - AttributeName: "PK"
            KeyType: "HASH"
          - AttributeName: "SK"
            KeyType: "RANGE"
        BillingMode: "PAY_PER_REQUEST"
        StreamSpecification:
          StreamViewType: 'NEW_IMAGE'
```

Let’s start from the bottom and work our way up.

In the `resources` block, I'm defining my DynamoDB table. With the Serverless Framework, you can write CloudFormation to provision resources as needed. See the "Provisioning" chapter of the Operations supplement for more on using CloudFormation. Notice that I'm configuring a DynamoDB Stream for my table by setting the `StreamSpecification` property.

Above the `resources` block is my `functions` block where I declare my Lambda functions. I have one function here—the `streamHandler` function—and it's triggered by a `stream` event from my DynamoDB stream.

Finally, at the top I have meta configuration about my project. In addition to having the region, stage, and runtime, I'm also declaring the IAM permissions for my function. Note that I have permissions relating to streams so that I can read from the DynamoDB Stream. Check out the 'Authorization' chapter of the Operations supplement for more details on IAM.

In my Lambda function, I'll receive a batch of records with the following format:

```
{
    "Records": [
        {
            "eventID": "7de3041dd709b024af6f29e4fa13d34c",
            "eventName": "INSERT",
            "eventVersion": "1.1",
            "eventSource": "aws:dynamodb",
            "awsRegion": "us-west-2",
            "dynamodb": {
                "ApproximateCreationDateTime": 1479499740,
                "Keys": {
                    "Timestamp": {
                        "S": "2016-11-18:12:09:36"
                    },
                    "Username": {
                        "S": "John Doe"
                    }
                },
                "NewImage": {
                    "Timestamp": {
                        "S": "2016-11-18:12:09:36"
                    },
                    "Message": {
                        "S": "This is a bark from the Woofer social network"
                    },
                    "Username": {
                        "S": "John Doe"
                    }
                },
                "SequenceNumber": "13021600000000001596893679",
                "SizeBytes": 112,
                "StreamViewType": "NEW_IMAGE"
            },
            "eventSourceARN": "arn:aws:dynamodb:us-east-
1:123456789012:table/BarkTable/stream/2016-11-16T20:42:48.104"
        }
    ]
}
```

Each record includes the type of operation (`INSERT`, `MODIFY`, `REMOVE`) as well as information on the primary key of the item affected. Depending on the type of stream I configured, it will also show the item before the operation, after the operation, or both before and after the operation. You will usually use the item after the operation in your analytics system.

5.4. Conclusion

In this chapter, we saw the final way to handle analytics with your DynamoDB table. For those that need more real-time updates, DynamoDB Streams are a great way to continuously export data from your table into a system for processing.

However, this is likely the hardest way to handle analytics with your DynamoDB table. If you have a small amount of data, you can get away with writing your data directly into a standard relational database and runnig analytics queries there. But if you have a lot of data, you'll need to figure out a way to ingest and update data into a large-scale analytics system that's not built for large amounts of small updates. We discussed some of those strategies here.